Higher Order Functions

Mattox Beckman
beckman@iit.edu

Illinois Institute of Technology

The purpose of this lecture is to give you an introduction to higher order functions. As a result of this lecture, you should. . .

- know how to create higher order functions
- understand the concept of *closures*
- know how to use and write the following kinds of higher order functions:
  - function combination — `twice`, `compose`
  - interface changes — `curry`, `uncurry`
  - list processing — `fold_right`, `map`, `zip_with`

A type is said to be *first class* type when it can be

- **assigned** to a variable, **passed** as a parameter, or **returned** as a result

Examples:

- APL: scalars, vectors, arrays
- C: scalars, pointers, structures
- C++: like C, but with classes
- Scheme, Lisp, ML: scalars, lists, tuples, functions

The Kind of Data a Program Manipulates Changes the Expressive Ability of a Program


```
1 # let double x = x * 2;;
2 val double : int -> int = <fun>
3 # let inc x = x + 1;;
4 val inc : int -> int = <fun>
5 # let compose f g = fun x -> f (g x);;
6 val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

- These are *higher order types*
- Important skill: be able to tell from the *type* of the function what action it performs.
- Try combining `compose`, `inc`, and `double`. What happens?

Twice

```
# let twice f x = f ( f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
# twice inc 2;;
- : int = 4
```

Can you write `twice` using `compose`?

Sectioning

- A function that can create another function is a HOF.
- The simplest way to make one is to pass in only some of the arguments a function requires.

  How can you interpret the type `int -> float -> string`?
- When this happens to operators, it's called *sectioning.*

```
# (+);;
- : int -> int -> int = <fun>
# let inc = (+) 1;;
val inc : int -> int = <fun>
```


Free Variables

- Consider what happens here....

```
# let plus x y = x + y;;
val plus : int -> int -> int = <fun>
# let inc = plus 1;;  (* no parameter for y *)
val inc : int -> int = <fun>
# inc 5;;
- : int = 6
# let x = 20;;
val x : int = 20
```

What will be the result of `inc 5` now?

Closures

- A *closure* is an expression (usually a function) along with an environment.

```
# let plus x y = x + y;;
val plus : int -> int -> int = <fun>
# let inc = plus 1;;  (* no parameter for y *)
val inc : int -> int = <fun>
```

$$\texttt{inc} = \triangleleft\{x \mapsto 1\}, \texttt{fun y => x + y} \triangleright$$

- The "local" environment will have precedence over any other variables that exist.
- Note that no computation is done at all until all arguments are received.

Two Isomorphic Types

- What is the difference between these two types?
  - `int -> int -> int`
  - `(int * int) -> int`

```
# let foo x y = x + y;;
val foo : int -> int -> int = <fun>
# let bar (x,y) = x + y;;
val bar : int * int -> int = <fun>
```

- What will happen if we try to section these functions?

Curried Functions

- Type `int -> int -> int` is said to be in *curried* form.
- What will the following function do? You are only given the type.

```
# let foo secret stuff
val foo : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

- Can you figure out how to write this function?

Curry and Uncurry

- Here are the function definitions. You should memorize them.

```
# let curry f a b = f (a,b);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f (a,b) = f a b;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fu
```

- Now we can section an uncurried function....

```
# let bar (x,y) = x + y;;
val bar : int * int -> int = <fun>
# let inc = (curry bar) 1;;
val inc : int -> int = <fun>
```


Flip

- We can also do other weird things....

```
# let flip f a b = f b a;;
val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
# let (-) = flip (-);;
val ( - ) : int -> int -> int = <fun>
# 2 - 5;;
- : int = 3
```

Combining Lists and Functions

- You can make *lists of functions*.
- Note that we don't have to know in advance how many functions we will be processing.
- Why isn't this type as general as `compose`?
  `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

```
# let rec complist flst x =
    match flst with
    | [] -> x
    | f::fs -> f (complist fs x);;
val complist : ('a -> 'a) list -> 'a -> 'a = <fun>
# complist [inc; double; inc] 4;;
- : int = 11
```


Repeating a function n times

- We do not have to know in advance how many times we will process them.

```
# let rec fnth n f x =
    match n with
    | 0 -> x
    | _ -> f (fnth (n-1) f x);;
val fnth : int -> ('a -> 'a) -> 'a -> 'a = <fun>
# fnth 5 inc 2;;
- : int = 7
# fnth 10 double 10;;
- : int = 10240
```


Mapping functions the hard way

Consider the following definitions. What do they have in common?

```
# let rec inclist lst = match lst with
    | [] -> []
    | x::xs -> x + 1 :: inclist xs;;
val inclist : int list -> int list = <fun>
# let rec doublelist lst = match lst with
    | [] -> []
    | x::xs -> x * 2 :: doublelist xs;;
val doublelist : int list -> int list = <fun>
# inclist [2;3;4];;
- : int list = [3; 4; 5]
# doublelist [2;3;4];;
- : int list = [4; 6; 8]
```


Mattox's Rule of Computing

The computer exists to work for us; not us for the computer. If you are doing something repetitive for the computer, you are doing something wrong.
Stop what you're doing and find out how to do it right.

Mapping functions the easy way

`map f` $[x_1; x_2; \ldots x_n] = [f(x_1); f(x_2); \ldots f(x_n)]$

```
# let rec map f lst = match lst with
    | [] -> []
    | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map inc [2;3;4];;
- : int list = [3; 4; 5]
# map double [2;3;4];;
- : int list = [4; 6; 8]
```

- `inc` and `double` have been transformed into recursive functions.
- How would you have done this in C++ or Java?

Related function: `zip`

```
# let rec zip f alst blst =
    match alst,blst with
    | [],_ -> []
    | _,[] -> []
    | a::aa, b::bb -> f a b :: zip f aa bb;;
val zip : ('a -> 'b -> 'c) -> 'a list -> 'b list
          -> 'c list = <fun>
# zip (+) [2;3;4] [5;6;8];;
- : int list = [7; 9; 12]
# zip (fun a b -> a * a + b * b) [2;3;4] [3;4;5];;
- : int list = [13; 25; 41]
```

- Side note: `zip` is sometimes known as `map2` or `zipwith`.

Folding functions

Consider the following definitions. What do they have in common?

```
# let rec sumlist lst = match lst with
    | [] -> 0
    | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# let rec prodlist lst = match lst with
    | [] -> 1
    | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# prodlist [2;3;4];;
- : int = 24
```


fold

`fold f` $[x_1; x_2; \ldots x_n]\ z = f(x_1, f(x_2, f(\ldots, z)))$

```
# let rec fold_right f lst z = match lst with
    | [] -> z
    | x::xs -> f x (fold_right f xs z);;
val fold_right : ('a -> 'b -> 'b) -> 'a list
                 -> 'b -> 'b = <fun>
# fold_right (+) [2;3;4] 0;;
- : int = 9
# fold_right (fun a b -> a * a + b) [2;3;4] 0;;
- : int = 29
```

- To use `fold`, we specify the function *and* the base case.

Encoding Recursion using `fold`

```
# let rec flatten lst = match lst with
    | [] -> []
    | x::xs -> x @ flatten xs;;
val flatten : 'a list list -> 'a list = <fun>
# flatten [ [2;3]; [3;4;5]; [6;0]];;
- : int list = [2; 3; 3; 4; 5; 6; 0]
```

Operation Recursive Call Base Case

```
let flatten lst =
    fold_right (fun x y -> x @ y) lst [];;
```

- **Note well**: the second parameter of the function argument to `fold_right` represents the *result* of the recursive call.

Another example

- You can write `map` using `fold`.

```
# let fmap f lst =
    fold_right (fun x y -> f x :: y) lst [];;
val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
# fmap inc [2;3;4];;
- : int list = [3; 4; 5]
```

- Note, the reverse direction will not work. Why not?

Problems

1. Write a function `flipuc` that flips uncurried functions. Do this using only `flip`, `curry`, and `uncurry`.
2. Write the function that has the type
   `('a -> 'b) -> 'a * 'c -> 'b`
3. Use `fold_right` to write a function that takes a list and then returns it.
4. Use `fold_right` to write a function that takes a list and removes all elements less than zero.

Problem 1

- Write a function `flipuc` that flips uncurried functions. Do this using only `flip`, `curry`, and `uncurry`.

```
# let flipuc f = uncurry (flip (curry f));;
val flipuc : ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>
# let sub (a,b) = a - b;;
val sub : int * int -> int = <fun>
# let sub = flipuc sub;;
val sub : int * int -> int = <fun>
# sub (5, 2);;
- : int = -3
```

Problem 2

- Write the function that has the type `('a -> 'b) -> 'a * 'c -> 'b`

```
# let pro1 f (x,y) = f x;;
val pro1 : ('a -> 'b) -> 'a * 'c -> 'b = <fun>
# pro1 inc (4,8);;
- : int = 5
# pro1 inc (4,"hi");;
- : int = 5
```


Problem 3

- Use `fold` to write a function that takes a list and then returns it.

```
# let cons a b  = a :: b;;
val cons : 'a -> 'a list -> 'a list = <fun>
# let return lst = fold_right cons lst [];;
val return : 'a list -> 'a list = <fun>
# return [2;3;4];;
- : int list = [2; 3; 4]
```

Yeah, it's pretty boring, but knowing the values that make the identity is important.

Problem 4

- Use `fold` to write a function that takes a list and removes all elements less than zero.

```
# let gtzero lst =
    fold_right (fun a b -> if a >= 0
                           then a :: b
                           else b) lst [];;
val gtzero : int list -> int list = <fun>
# gtzero [2;-3;4;-65];;
- : int list = [2; 4]
```

A related function `filter` does the same thing, but takes a parameter for the `a >= 0` part.